

# 5

## Stack ADT

### 목적

이번 실습에서 여러분은

- 두 가지 Stack ADT를 구현한다. 한 가지는 스택의 배열을 이용한 표현이고, 다른 하나는 단순 연결 리스트를 이용한 표현이다.
- 후위 표기 형태의 산술적 표현을 계산하는 프로그램을 작성한다.
- 주어진 복제 생성자(copy constructor)의 제한 사항을 분석하고, 스택의 단순 연결 리스트적 표현을 위한 개선된 복제 생성자를 작성한다.
- 스택을 이용하여 만들 수 있는 순열들의 종류를 분석한다.

### 개요

선형 자료 구조를 사용하는 많은 응용들은 List ADT에서 제공하는 모든 기능이 필요하지 않다. 이러한 응용들을 List ADT를 이용하여 작성한다 할지라도, 작성된 프로그램은 때에 따라서는 비효율적일 수 있다. 이러한 문제를 해결하기 위한 방법 중 한 가지는 더 강제적인 기능들의 집합들을 지원하는 새로운 선형 자료 구조를 정의하는 것이다. 이러한 ADT들을 정의함으로써, 다양한 응용들에 필요한 ADT들을 생성하고, List ADT보다 더 효율적이고, 쉽게 적용할 수 있는 ADT들을 만들 수 있다.

스택은 선형 자료 구조에 제한을 추가한 하나의 예이다. 스택의 원소들은 가장 최

근에 추가된 원소인 **top**에서 가장 먼저 추가된 원소인 **bottom**까지 순서화되어 있다. 모든 삽입과 삭제는 스택의 **top**에서 실행된다. **push**연산은 하나의 원소를 Stack의 **top**에 추가하며 최상위의 Stack 원소를 제거하기 위해서는 **pop**연산을 사용한다. 아래의 표는 **push**와 **pop**연산의 순서를 보인 것이다.

<i>Push a</i>	<i>Push b</i>	<i>Push c</i>	<i>Pop</i>	<i>Pop</i>
		<i>c</i>		
	<i>b</i>	<i>b</i>	<i>b</i>	
<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
---	---	---	---	---

삽입과 삭제의 이러한 제한들은 스택의 특성인 "*last in, first out*" 행위를 표현한다. 비록 스택 자료구조를 간단하게 정의하였으나, 스택은 대부분의 컴퓨터 구조의 기본적인 요소인 원시적인 스택을 지원하는 시스템 소프트웨어에서 폭 넓게 사용되어진다.

## Stack ADT

### 원소

스택의 원소들은 유전적인 형 SE이다.

### 구조

스택 원소들은 가장 최근에 추가된 원소(top)에서 가장 먼저 추가된 원소(bottom)로 선형적으로 순서화되어 있다. 원소들은 (push)로 삽입하고 스택의 맨위(top)에서 (pop)으로 제거한다.

### 기능

**Stack** (int maxNumber = defMaxStackSize)

요구사항:

없음.

결과:

생성자. 하나의 빈 스택을 생성한다. 필요시, maxNumber 원소를 포함하는 스택을 위해 충분한 메모리를 할당하여라.

**~Stack ( )****요구사항:**

없음.

**결과:**

과피자. 하나의 스택을 저장하기 위해 사용된 메모리를 할당 취소한다.

**void push (const SE &newElement)****요구사항:**

스택이 가득 차 있지 않을 경우.

**결과:**

newElement를 stack의 윗 부분에 넣는다.

**SE pop ( )****요구사항:**

스택이 비어 있지 않을 경우.

**결과:**

스택에서 가장 최근에 추가된(top) 원소를 제거한다.

**void clear ( )****요구사항:**

없음.

**결과:**

스택에서 모든 원소를 제거한다.

**int empty ( ) const****요구사항:**

없음.

**결과:**

스택이 비어있으면, 1을 반환하고 이외에는 0을 반환.

**int full ( ) const**

요구사항:

없음

결과:

스택이 가득 차 있으면, 1을 반환하고 이외에는 0을 반환.

**void showStructure ( ) const**

요구사항:

없음.

결과:

스택에서 하나의 원소를 출력한다. 만약 스택이 비어 있으면, "Empty stack"을 출력한다. 이 기능은 시험/디버깅 목적을 위해서만 사용된다. 그리고 이 기능은 stack의 원소가 C++의 이미 정의된 데이터 형(int, char 등)중의 하나인 경우에만 지원한다.

---

## 실습 5 : 표지

날짜 \_\_\_\_\_ 구분 \_\_\_\_\_

성명 \_\_\_\_\_

강사가 여러분에게 할당한 다음에 연습문제들에 할당된 열에 체크표시를 하고 이 겹장을 앞으로의 연구에서 여러분이 제출할 과제물 앞에 붙이시요.


	할 일	완 료
실습 전 연습	✓	
연 결 연습	✓	
실습 중 연습 1		
실습 중 연습 2		
실습 중 연습 3		
실습 후 연습 1		
실습 후 연습 2		
	총 점	

## 실습 5 : 실습 전 연습

날짜 \_\_\_\_\_ 구분 \_\_\_\_\_  
 성명 \_\_\_\_\_

다양한 오퍼레이팅 환경에서 효율적으로 ADT를 사용하려면 여러 개의 ADT를 구현해야 한다. 하드웨어와 응용 프로그램에 따라서 ADT 오퍼레이션의 일부 또는 전체에 대한 실행 시간을 감소시키는 구현을 요구하기도 하거나 ADT 원소들을 저장하기 위하여 사용된 메모리의 양을 감소시키는 구현을 요구하기도 한다. 예제에서 여러분은 두 개의 Stack ADT를 구현해야 한다. 첫 번째 구현은 배열에 스택을 저장하는 것이다. 다른 하나는 각각의 스택에 원소를 분리 저장하고 이것들을 하나의 스택으로 구성하기 위해서 모든 원소들을 연결한다.

**1단계 :** 스택 원소들을 저장하기 위해서 배열을 사용하는 Stack ADT의 오퍼레이션을 구현하라. 스택의 크기는 가변적이다. 그러므로 우리는 스택이 가질 수 있는 스택의 최대 원소의 수를 저장하기 위해 **MaxSize**와 스택에서 가장 윗 부분 원소의 표시를 나타내는 **top**과 스택 원소 자체를 나타내는 **element**를 필요로 한다. 파일

 *stackarr.h*상에 여러분의 구현을 기술하여라. 그리고 파일 *show.cpp*에 *showstructure* 오퍼레이션의 구현하여라. 만약 여러분이 *template*의 사용이 친숙하지 않다면, 63-65 페이지의 제 4장에 대한 내용을 참고하여라.

```
const int defMaxStackSize = 10; // Default maximum stack size
template <class SE>
class Stack
{
public:
    // Constructor
    Stack (int maxNumber = defMaxStackSize);
    // Destructor
    ~Stack ( );
    // Stack manipulation operations
    void push (const SE &newElement); // Push element
    SE pop ( );                       // Pop element
    void clear ( );                   // Clear stack
}
```



```

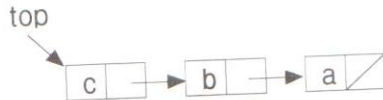
// Stack status operations
int empty ( ) const;           // Stack is empty
int full ( ) const;           // Stack is full
// Output the stack structure -- used in
// testing/debugging
void showStructure ( ) const;

private:
    // Data members
    int maxSize, // Maximum number of elements in the tack
        top;    // Index of the top element
    SE *element; // Array containing the stack elements
};

```

**2단계 :** Stack ADT에 관한 여러분의 배열 구현을 파일 `stackarr.cpp`에 저장하여라. 여러분의 코드에 주석을 확실히 하여라.

Stack ADT에 관한 여러분의 배열 구현에서 여러분은 Stack을 선언하였을 때 스택을 저장하기 위해서 사용된 메모리를 할당한다. 최후의 배열은 특별한 응용 프로그램에서 필요로 하는 가장 커다란 스택도 수용할 수 있도록 충분히 커야 한다. 불행히도 스택 배열에 있어 대부분의 시간은 실질적으로 커다란 것은 아니고, 보조 메모리는 사용하지 않을 것이다.



위의 방식과는 달리 접근하는 방식은 새로운 원소를 스택에 추가할 때마다 원소 단위로 메모리를 할당하는 것이다. 이 방법으로 여러분은 여러분이 메모리가 실질적으로 필요할 때만 메모리를 할당하게 된다. 모든 시간상에서 메모리를 할당하기 때문에 원소들은 메모리 위치에 대한 인접 집합을 차지하지 않는다. 결과적으로 stack의 표현은 아래의 그림과 같이 연결 리스트(linked list)를 형성하기 위해 모든 원소를 연결한다.

Stack ADT에 관한 연결 리스트 구현을 생성하는 것은 배열 구현을 개발하였던 것보다 더 프로그래밍 작업에 적절하기 때문이다. 이 작업을 단순화하는 유일한 방법은 두 개의 template 클래스로 구현을 나누는 것이다. 하나의 클래스는 전체적인 스택 구조에 초점을 두고, 또 다른 하나는 StackNode 클래스라 하여 연결 리스트의 각각의 노드에 초점을 두는 것이다.

StackNode 클래스에서부터 시작하자. 연결 리스트의 각 노드들은 스택 원소와 연결 리스트의 다음 원소를 포함하는 노드로의 포인터를 포함한다. StackNode 클래스에 의해서 주어지는 유일한 기능은 설명된 노드를 생성하는 생성자(constructor)이다.

StackNode 클래스로의 접근은 Stack 클래스의 멤버(member) 함수에서 제한한다. StackNode의 모든 member들을 프라이빗(private)로 선언함으로써 직접적인 연결 리스트 노드들의 참조로부터 다른 클래스를 보호한다. Stack을 StackNode의 friend로 선언함으로써 StackNode의 멤버들을 Stack 클래스가 접근 가능하도록 만든다. 파일의 *stacklnk.h*에 다음과 같은 클래스 선언으로 위에서 기술한 성질들을 반영한다.

```
template <class SE>
class StackNode    // Facilitator class for the Stack class
{
private:
    // Constructor
    StackNode (const SE &elem, StackNode *nextPtr);
    // Data members
    SE element;           // Stack element
    StackNode *next;      // Pointer to the next element
    friend class Stack<SE>;
};
```

StackNode 클래스 생성자는 Stack에 노드를 추가하는데 사용한다. 예를 들어 다음의 선언은 문자들의 스택에 'd'를 포함하는 노드를 추가한다. 이 예제 template 매개변수 SE는 type char와 동등해야 하고, top은 type StackNode\*이다.

```
top = new StackNode<SE> ('d', top);
```

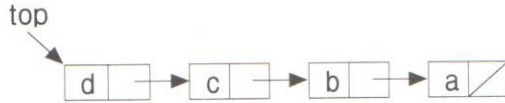
new 연산자는 연결 리스트 노드에 메모리를 할당하고 StackNode 생성자라 부른다. 이것은 원소를 추가('d')하고 리스트의 다음 노드를 (top) 포인터하는 두 가지를 수행한다.



마침내, 할당 연산자는 새롭게 할당된 노드의 주소를 top으로 할당하고, 그렇게 함



으로써 노드에 대한 생성과 연결이 완료된다.



Stack 클래스의 멤버 함수는 Stack ADT에서의 기능들을 구현한다. 연결 리스트의 처음 부분, 또는 스택의 top에 있는 노드에 포인터를 동등하게 유지한다. 다음과 같은 Stack 클래스들을 *stacklnk.h* 파일에 넣는다.


```

template <class SE>
class Stack
{
public:
    // Constructor
    Stack (int ignored = 0);
    // Destructor
    ~Stack ( );
    // Stack manipulation operations
    void push (const SE &newElement); // Push element
    SE pop ( ); // Pop element
    void clear ( ); // Clear stack
    // Stack status operations
    int empty ( ) const; // Stack is empty
    int full ( ) const; // Stack is full
    // Output the stack structure -- used in
    // testing/debugging
    void showStructure ( ) const;
    // In-lab operation
    Stack (const Stack &valueStack); // Copy constructor
private:
    // Data member
    StackNode<SE> *top; // Pointer to the top element
};
  
```

**3단계 :** 스택 원소를 저장하기 위해서 단순 연결 리스트를 사용하는 Stack ADT에서의 기능을 구현하라. 연결 리스트의 각 노드들은 stack 원소와 스택에서 다음 원소를 포함하는 노드의 포인터(next)를 포함한다. 여러분의 구현에는 또한 스택의 최상위 원소를 포함하는 노드의 포인터(top)을 유지해야 한다. 여러분의 구현은 파일

*stacklnk.h*의 클래스 선언을 기본으로 한다. 파일 *show5.cpp*으로 ShowStructure 기능

을 구현한다.


 4단계 : 여러분의 Stack ADT에 관한 연결 리스트 구현을 파일 `stacklnk.cpp`에 저장하고 여러분의 코드에 대한 주석도 기입하여라.

## 실습 5 : 연결 연습

이름 \_\_\_\_\_ 구분 \_\_\_\_\_

성명 \_\_\_\_\_

여러분의 강사와 함께 이 연습을 수업 이전이나 수업 동안에 완료했는지를 검토하라.

 파일 `test5.cpp`에 있는 시험 프로그램은 아래에 설명한 명령에 따라 Stack ADT에 대한 여러분의 구현을 대화적으로 검사하는 프로그램이다.

명령어	동 작
+x	원소 x를 스택의 top에 넣는다. (push)
-	top 원소를 빼내고(pop) 출력한다.
E	스택의 empty 여부를 보고한다.
F	스택의 full 여부를 보고한다.
C	스택을 초기화한다.
Q	시험 프로그램을 종료한다.

**1단계 :** 시험 프로그램을 컴파일하고 링크하여라. 여기서 프로그램을 컴파일한다는 것은 문자 스택을 위해 배열 구현을 생성하는(`stacklnk.cpp` 파일) Stack ADT에 관한 여러분의 배열 구현을 컴파일하는 것임을 주목하여라.

**2단계 :** 아래와 같은 시험 항목을 추가하여 시험 계획을 완성하여라.

- 단지 하나의 원소만을 가지고 있는 스택에서 하나의 원소를 빼낸다. (pop)
- pop의 연속 수행에 의해서 비어 있는 스택에서 한 원소를 넣는다. (push)
- 결함이 있는 스택에서 한 원소를 빼낸다. (pop)
- 스택을 초기화한다.

**3단계 :** 시험 계획을 실행하여라. 만약 Stack ADT에 관한 배열 구현에 잘못이 발견되면, 수정한 후 시험 계획을 다시 실행하여라.

**4단계 :** 시험 프로그램을 수정하여 배열 구현이 파일 `stacklnk.cpp` 안에 있는 Stack ADT에 대한 연결 리스트 구현을 포함한다.

**5단계** : 시험 프로그램을 다시 컴파일하고 다시 링크하여라. 프로그램을 다시 컴파일하는 것은 문자 스택에 사용될 연결 리스트 구현을 생성하기 위해 Stack ADT의 연결 리스트 구현을 (파일 *stacklnk.cpp*) 컴파일하는 것이다.

**6단계** : Stack ADT에 관한 여러분의 연결 리스트 구현을 검사하기 위해 시험 계획을 사용하여라. 만약 여러분의 구현에 잘못이 있다면, 수정하고 시험 계획을 다시 실행하여라.

Stack ADT내의 Operation 시험 계획			
시험 항목	명 령	기대되는 결과	검사
순차적인 push	+a +b +c +d	a b c d	
순차적인 pop	- - -	a	
더 많은 push	+e +f	a e f	
더 많은 pop	- -	a	
비었나? 꽉찼나?	E F	0 0	
stack이 비었다	- -	빈 스택	
비었나? 꽉찼나?	E F	1 0	

※ 최상위 원소는 굵은 체로 표현

## 실습 5 : 실습 중 연습 1

날짜 \_\_\_\_\_ 구분 \_\_\_\_\_  
 성명 \_\_\_\_\_

우리는 일반적으로 중위(infix) 형태를 사용해 산술 수식을 작성한다. 이것은 아래와 같이 피연산자(operand) 사이에 연산자(operator)를 넣는 형태이다.

$$(3 + 4) * (5 / 2)$$

이러한 형태가 우리에게 매우 편리한 표기법이기는 하지만, 중위(infix) 형태는 계산되어지는 연산자의 순서를 표시하기 위해 삽입구를 사용해야 한다는 단점을 가지고 있다. 순서적인 삽입구는 계산 과정을 매우 복잡하게 한다.

여러분이 만약 왼쪽에서 오른쪽으로 간단히 연산자를 계산한다면 전개가 매우 간단하다. 그렇지만, 불행히도 전개 방법은 산술 수식의 중위(infix) 형태로는 실행할 수가 없다. 그러나 수식이 후위(postfix) 형태라면 실행할 수가 있다. 산술 수식이 후위 형태라면 각 연산자를 연산수 후에 바로 대체할 수 있다. 위의 수식을 후위 형태로 서술하면 다음과 같다.

$$34 + 52 / *$$

두 형태 모두 같은 순서(왼쪽에서 오른쪽으로 읽기)로 숫자가 위치하는 것에 주목하라. 그러나 연산자의 순서는 다르다. 왜냐하면, 후위 형태에서 연산자들은 그들의 계산되어지는 순서대로 위치한다. 결과적으로 후위 수식은 처음에 읽기는 어렵다. 그러나 계산하기는 쉽다. 우리가 필요로 하는 모든 것은 결과를 바로 스택에 위치시키는 것이다.

후위 형태의 산술 수식에 사용되는 수식은 한자리의 수이며, 양의 정수이고 4개의 기본 연산자(덧셈, 뺄셈, 나눗셈, 곱셈)로 이루어진 산술 수식만을 가정한다. 이러한 수식은 부동 소수점 수의 스택을 이용하여 다음의 알고리즘을 이용하여 계산할 수 있다.

문자 단위로 수식을 읽어라.

만약 그 문자가 한자리 수('0'에서 '9'까지의 문자)와 대응한다면, 그 대응하는 부동 소수점 수를 스택에 넣는다.(push)



만약 그 문자가 산술 연산자들('+', '-', '\*', 그리고 '/' 문자)중의 하나와 대응된다면,

- 스택에서 하나의 숫자를 빼낸다. 그것을 *operand1*이라 한다.(pop)
- 스택에서 하나의 숫자를 빼낸다. 그것을 *operand2*이라 한다.(pop)
- 다음과 같이 산술 연산자를 사용하여 이들 연산수를 결합한다.

Result = operand2 operator operand1

- Result을 스택에 넣는다.

수식의 끝에 도달했을 때, 스택에서 마지막 남아있는 수를 빼낸다.(pop) 이 수는 수식의 값이다.

다음의 산술 수식에 이 알고리즘을 적용하여라.

34 + 52 / \*

다음과 같은 계산이 이루어진다.

```
'3' : Push 3.0
'4' : Push 4.0
'+' : Pop operand1 = 4.0
      Pop operand2 = 3.0
      Combine, Result = 3.0 + 4.0 = 7.0
      Push 7.0
'5' : Push 5.0
'2' : Push 2.0
'/' : Pop operand1 = 2.0
      Pop operand2 = 5.0
      Combine, Result = 5.0 / 2.0 = 2.5
      Push 2.5
'*' : Pop operand1 = 2.5
      Pop operand2 = 7.0
      Combine, Result = 7.0 * 2.5 = 17.5
      Push 17.5
'\n' : Pop, Value of expression = 17.5
```

**1단계 :** 산술 수식의 후위 형태를 읽고 계산하고 값을 출력하는 프로그램을 작성하여라. 그 수식은 한자리의 양의 정수('0'에서 '9'까지의 수)와 네 개의 기본적인 연산자('+', '-', '/', '\*')로 구성된 수식으로 가정하자. 또한 그 수식은 한 줄로 구성되며

키보드로부터 입력된 수식으로 가정한다.

**2단계** : 각 산술 수식에 기대되는 결과값으로 채워지도록 다음의 시험 계획을 완성 하여라. 그리고 이 시험 계획에 추가적인 산술 수식을 포함할 수 있도록 한다.

**3단계** : 시험 계획을 실행하라. 실행 중 잘못을 발견하게 된다면, 잘못된 것을 수정 하고, 다시 실행하여라.

후위 산술 수식 계산 프로그램을 위한 시험 계획			
시험 항목	산술 수식	기대되는 결과	검사
연산자 하나	$34+$		
내포된 연산자	$34+52/*$		
한결같지 않은 내포	$93*2+1-$		
끝에 있는 모든 연산자	$4675-+*$		
'0'으로 나누기	$02/$		
한자리 숫자	$7$		